Scalable P2P Reconciliation Infrastructure for Collaborative Text Editing

Mounir Tlili, Reza Akbarinia
INRIA and LINA
Nantes, France
Mounir.Tlili@univ-nantes.fr, Reza.Akbarinia@inria.fr

Esther Pacitti, Patrick valduriez
INRIA and LIRMM
Montpellier, France
pacitti@lirmm.fr, Patrick.Valduriez@inria.fr

***Abstract*—We address the problem of optimistic replication for collaborative text editing in Peer-to-Peer (P2P) systems. This problem is challenging because of concurrent updating at multiple peers and dynamic behavior of peers. Operational transformation (OT) is a typical approach used for handling optimistic replication in the context of distributed text editing. However, most of OT solutions are neither scalable nor suited for P2P networks due to the dynamic behavior of peers. In this paper, we propose a scalable P2P reconciliation infrastructure for OT that assures eventual consistency and liveness despite dynamicity and failures. We propose a P2P logging and timestamping service called P2P-LTR (P2P Logging and Timestamping for Reconciliation) which exploits a distributed hash table (DHT) for reconciliation. While updating replica copies at collaborating peer editors, updates are stored in a highly available P2P log. To enforce eventual consistency, these updates must be retrieved in a specific total order to be reconciled at the peer editors. P2P-LTR provides an efficient mechanism for determining the total order of updates. It also deals with the case of peers that may join and leave the system during the update operation. We evaluated the performance of P2P-LTR through simulation; the results show the efficiency and the scalability of our solution.**

Keywords- Optimistic replication; reconciliation; P2P systems; distributed hash tables; collaborative text editing

I. INTRODUCTION

Collaborative applications are getting common as a result of rapid progress in distributed technologies like the Web 2.0. Building these applications on top of P2P networks has many advantages: decentralization, self-organization, scalability and fault-tolerance. One of the main types of collaborative applications is collaborative text editing, e.g. the wikis that are actually very popular editors. Generally, the wiki systems are built over the traditional centralized architecture where the whole set of pages of a wiki reside on a single server. Consequently, in the case of failure or offline work, data are unavailable. Moreover, the system does not scale easily. This is why recently there have been some efforts to shift the wikis to fully decentralized systems relying on P2P networks, e.g. XWiki [27] that is a second generation wiki working over a P2P network. However, providing data consistency in the presence of concurrent updates on the same data is quite challenging in these systems due to the dynamicity of peers and the scalability and fault-tolerance requirements.

In a P2P text editing application, users need to work on shared documents even though they are disconnected from the network, e.g. in a train or another environment that does not provide good network connection. This requires that users hold local replicas of shared documents. This is a typical case of collaborative applications that requires optimistic replication to assure data availability at anytime. Optimistic replication [21] is a well known and efficient solution for multimaster replication. It allows asynchronous updating of replicas so that applications can progress even though some nodes are disconnected. This enables asynchronous collaboration among users. However, concurrent updates may cause replica divergence and conflicts, which should be reconciled to enforce that replicas converge to the same state, property known as *eventual consistency*. In this paper, we propose a new infrastructure for P2P timestamping and reconciliation, that may be useful for different types of P2P collaborative applications, however to be able to express the functionalities of our infrastructure, we focus on P2P collaborative text editing applications.

Operational transformation (OT) [14] is one of the main distributed frameworks used for handling optimistic replication in the context of distributed text editing applications. The two most important aspects of OTs is that they are generic, i.e. they can handle different types of data (text, multimedia, etc), and it uses a set of generic operation transforms for reconciliation. However, to ensure eventual consistency during reconciliation, all involved operations must be stamped in *continuous total order* (timestamps values are integer ascending order). One well known OT solution [14] uses a centralized timestamp server to stamp operations (update, insert…) which limits scalability and may block in case of failures, or introduce bottlenecks. Besides, the size of the log used by OT at the timestamp server tends to be hudge and may be unmanageable in a single node. P2P solutions for text editing [2][13][16] are not generic as OT or are quite complex in practice.

Thus, an important challenge in P2P text editing applications is to provide a complete P2P solution facing the following problems: eventual consistency management, decentralized timestamper service, large scale and consistent log management. In this paper, we propose a new scalable

P2P logging and timestamping infrastructure called P2P-LTR (Logging and Timestamping for Reconciliation). In our solution, replica copies are optimistically updated at collaborating *peer editors*, even if disconnected. To enforce eventual consistency, a necessary condition is that updates must be applied in a continuous total ordered during reconciliation [21] at each peer editor. Whenever a timestamp value is established and valid for an update, then the timestamped update is stored in a P2P Log, in our case over a distributed hash table (DHT), to be available for the other peer editors. Thus, using the P2P Log, each *peer editor* may retrieve concurrent updates in total order to be able to reconcile locally.

Providing total order despite failures and dynamic behavior of peers is one of the major challenges of P2P-LTR which we address in this paper. P2P-LTR infrastructure provides the following capabilities: (a) a scalable P2P continuous timestamp protocol for managing timestamps in a DHT that assures eventual consistency and liveness; (b) a highly available log (P2P-Log) for storing timestamped updates; and (c) a retrieval mechanism that retrieves the timestamped updates in timestamp order to assure *eventual consistency* [21]. To validate P2P-LTR, we implemented it using two popular DHTs: OpenChord [15] and Free Pastry [8]. Moreover, we simulated our algorithms, and the performance evaluation results show the efficiency and scalability of our solution.

Our work has been done in the context of the XWiki Concerto project [27] for collaborative P2P editing. The rest of this paper is organized as follows. In Section 2, we present the main terms and concepts used in this paper, and we give the problem statement. In Section 3, we describe P2P-LTR's model and its main algorithms. In Section 4, we propose a complete solution for dealing with peers' dynamic behavior. Section 5 presents our performance evaluation. In Section 6, we discuss related work. Section 7 concludes.

II. Main Concepts and Problem Definition

In this section, we first introduce some terms and concepts that constitute the basis of our work. Then, we state the problem we address in this paper.

A. Main Concepts and Assumptions

A *replica* is a copy of a document D. We assume optimistic multimaster replication, i.e. multiple primary copies of a document D, noted $D_1, D_2, \ldots, D_n$, are stored at different peers for reading and writing. Most of the optimistic replication solutions [21] *assures eventual consistency* among replicas, i.e. if all users stop updating the replicas (e.g. after a time t), eventually all replicas converge to the same state [21].

P2P-LTR manages the operations in the form of patches. A *patch* is the unit of reconciliation that corresponds to a sequence of update operations (insert, update, delete) generated when saving a document in a text editor. Once these patches are stamped in a *continuous timestamp order*, they are used for reconciliation by the integration algorithm of OT solution based on the transformation functions proposed by [14] for text reconciliation.

To be able to express the functionalities and properties of our infrastructure we focus on P2P collaborative text editing using OTs.

For reconciliation management, P2P-LTR timestamps the patches and enforces the *continuous timestamp order* property: for any two timestamps ts_1 and ts_2 generated for a document respectively at times t_1 and t_2, if $t_1 < t_2$ then we have $ts_1 < ts_2$ and for two consecutives timestamps values, say ts_1 and ts_2, $ts_2 = ts_1 + 1$. Notice that if $ts_i < ts_j$ then we say that ts_i *is previous wrt to* ts_j or we can alternatively say that the update operation timestamped with ts_i is *previous wrt* to the patch timestamped with ts_j.

P2P-LTR is built over a DHT (distributed hash table) overlay [20][23]: peer identifiers are chosen from an identifier space. Based on these identifiers, data placement is typically determined by a hash function which maps data identifiers to peer identifiers. That is, every patch (or any type of object) receives an unique identifier or key, and the peer with the identifier closest to the patch key is responsible for storing the key and its associate patch using the *put(key, patch)* primitive. When a client looks for a patch using the *get(key)* primitive, it contacts any peer in the DHT and the request is routed through the DHT ring until the peer with the identifier closest to the patch key is found. For simplicity, we assume the reliability of the DHT.

A *tentative patch* corresponds to a patch whose timestamp value is not assured to be in continuous timestamp order with respect to concurrent tentative patches on the same document. A *tentative patch* is *valid* if and only if its timestamp value is in continuous timestamp order. It can then be safely used by a set of transformation functions to perform reconciliation.

Our network model is semi-synchronous (falls between asynchronous and synchronous model), similar to the ones proposed in [1][12][13] and we assume no network partitions.

B. Problem Definition

The main challenges in designing a P2P infrastructure for collaborative text editing is to decentralize reconciliation assuring: (a) eventual consistency (imposes that patches are valid) despite dynamicity and failures (b) scalability, to support massive optimistic multi master replication (c) a reliable distributed log, which is used for reconciliation (d) and liveness (non blocking protocols).

III. P2P-LTR Capabilities

In this section, we first present the P2P-LTR model with respect to how peers collaborate to handle patch validation. Then, we present our algorithm used to validate patches. Finally, we describe how P2P-LTR ensures eventual consistency among replicas.

A. P2P-LTR Model

P2P-LTR is built over a DHT network. DHTs [20][23] provide a scalable solution for data location and lookup in large-scale P2P systems. All DHT systems have a simple distributed storage interface. Currently our solution is built over a DHT system based on Chord [23] or Pastry [20].

In P2P-LTR, each peer of the DHT may have four different roles:

- **User Peer**: implements the user application (denoted by *u*) that holds primary copies (in our case, documents). When a tentative patch on *D* is captured at *u* it is afterwards timestamped in continuous timestamp order. To accomplish this, the involved user peer must interact with a peer responsible for generating timestamp for *D*, called Master-key. By taking into account concurrent updates on *D*, a user peer may need to retrieve previous published patches at the Log-peers.
- **Log-Peer:** peer that is responsible for holding a valid timestamped patch *w.r.t* to a document.
- **Master-key Peer:** responsible for generating continuous timestamps for a document *D*. Each document is identified by a key value by hashing the document name. Using this key, the user peer locates the Master-key using a specific hash function h_{ts}. When a tentative patch is validated, the Master-key publishes the timestamped patch in the P2P-Log at specific Log-Peers. For this, the Master-key peers must provide a set of pairwise independent hash functions $H_r = \{h_1, h_2, \ldots, h_n\}$ which we call replication hash functions, used for implementing patch replication in the DHT. For a given key, the Master-key peer assumes the responsibility of sustaining the last timestamp value (denoted by *last_ts*) and mediating between concurrent updates.
- **Master-key Succ:** holds a copy of *last_ts* and replaces the Master-key in case of failures.

B. P2P-LTR Algorithm

In our model, each user peer has a local primary copy of the document. When u_1 updates a specific document *D*, the generated patch is considered as a tentative patch because its timestamp number is still not validated. The validation procedure consists of providing a *continuous timestamp value* to the new patch considering concurrent updates on the same document *D*, performed by other user peers (master of the same document). Since updates may be done concurrently, it may happen that a user generates new tentative patch without knowing that previous validated patches on the same document *D* are available at the P2P-Log. The patch timestamp validation procedure is done by contacting the Master-key of *D*.

To handle validation, at each user peer, each document has an associated local timestamp value (denoted by *ts*). Recall that the Master-key holds the last timestamp (denoted by *last_ts*) provided for the same document.

Thus, for a given document *D*, user peer u_1 first contacts the corresponding Master-key and asks it to publish the patch with the timestamp value *ts* (Lines 1-3, Algorithm 1). The Master-key of the document D is determined by applying a hash function h_t on *D*, and is located by using the lookup service of the DHT. If the Master-key's local timestamp value (*last_ts*) is equal to *ts*, then the Master-key increments by one *last_ts* value, and confirms the user peer u_1 that it will trigger the patch replication procedure. Then, the Master-key replicates the patch in the P2P-Log (at the Log-Peers) and acknowledges u_1, with a message containing the validated timestamp value (Lines 16-22, Algorithm 1).

In order to avoid having a big difference between the replicas on which the users are working, the users periodically check for new validated patches published by other users, e.g. each δ time units. If such patches exist they must be retrieved and locally integrated (Lines 6-7, Algorithm 1).

If the Master-key local timestamp value (*last_ts*) is greater than *ts*, this means that there are previous validated patches, which are generated by other users and must be integrated in u_1's document *D*. To accomplish this, u_1 must perform the retrieval procedure to get all missing patches in continuous timestamp order. Afterwards, u_1 restarts the timestamp validation procedure again until *last_ts* value is equal to *ts* value (see Lines 8-13, Algorithm 1).

To manage concurrent patch timestamp validation on the same document, the corresponding Master-key serves each user peer sequentially. That is, a new timestamp *ts* value for a given document *D* is provided after the replication of the previous timestamped (*ts-1*) patch on *D*.

Algorithm 1: p.publishPatch$_{ht}$(key, patch, ts)

```
                    User Application
1: //Patch tentative publication in the DHT
2: Master-Key-Peer.sendToPublish(key, patch, ts);
3: //Wait for successful publish ack from Master peer
4: If receive ack then
5:    Notify p of successful publish patch in DHT;
6: //Wait for a period of time δ before synchronization
7:    Synchronization;
8: Else
9:    Get the Missing Patchs from the DHT;
10:   Perform these Patches locally using Recon engine;
11:   Apply patches;
12:   Send patches;
13: end;
                    Master-key-Peer
14: sendToPublish(key, patch, ts)
15: begin
16:   If ts = last_ts then
17:         last_ts = last_ts + 1;
18:         For each h_i ∈ H do in parallel
19:               put(h_i(key, last_ts), patch);
20:         Replicate current-ts at master-succ;
21:         Send ack message, contain last_ts, to p;
22:         //indicating successful publish
23:   Else
24:         Send fail message to p;
25:         // indicating get missing patch
26:   end;
27: end;
```

Figure 1. P2P-LTR publish Patch process

C. *Eventual Consistency*

In this section, we illustrate how eventual consistency is guaranteed by P2P-LTR. For this, we show that if all users stop publishing patches (e.g. after a time t), eventually all replicas converge to the same state. Therefore, any new user who joins the system will see the same state of the document reflecting all modifications published before t.

Assume there are k users maintaining a replica of a shared document D, and u_1 be the user who has committed the last update. We show that if there are no new updates then all replicas converge to the same state as that of u_1. Let u_2 be a user that maintains a replica of document D. The user u_2 has one of the following situations: 1) after t, at least once he has got disconnected from the system; 2) after t, always he has been connected to the system. For each of these cases, we show that the document of u_2 eventually converges to that of u_1.

Let us first discuss the first case. In this case, when u_2 reconnects, he calls the *last-ts(key)* operation. Then, he compares the last timestamp *last_ts* generated by the Master-key peer with its local timestamp value *ts*. If *last_ts* is equal to *ts*, it means that there have been no committed patches during the absence of u_2. Otherwise (i.e. $last\_ts \neq ts$), u_2 retrieves all missing patches in continuous timestamp order and integrates them to its local document by using transformational approach. Thus, the state of its document becomes equal to that of u_1.

For the second case, i.e. u_2 has been always connected, recall that the users periodically check for the existence of missing patches in the DHT, i.e. each δ time units. If such patches exist, they are retrieved and locally integrated to the local replica at most after a time $t' \leq \delta + t$. Thus the document of u_2 eventually converges to that of u_1.

IV. Dealing With Peers' Dynamic Behavior In P2P-LTR

One of the main characteristics of P2P systems is the dynamic behavior of peers. In this section, we discuss how P2P-LTR deals with this P2P dynamic behavior. First, we consider the cases where a new Master-key peer joins or the current one leaves the system normally, i.e. without failing, and propose strategies by which P2P-LTR deals with these cases. Then, we address the situations where the Master-key peer failure.

A. *Master-key Peer's join and leave*

The Master-key peer of a key k can dynamically change due to the join/leave of peers to/from the DHT.

1) *Join*

This scenario focuses on the cases where a new peer joins the system and becomes Master-key for certain keys. In these cases, the joining peer triggers the DHT's stabilization procedure, which is used to keep nodes' successor pointers up to date in the DHT and to maintain the routing tables of all nodes, which is sufficient to guarantee correctness of lookups [23]. P2P-LTR sets up an event interrupter in the DHT's stabilization procedure. Using the event interrupter, P2P-LTR can get aware of the modifications in the topology of the DHT network. When the stabilization procedure is invoked, the event interrupter calls a P2P-LTR's function that is responsible for determining the new Master-key peer and its successor. Therefore, our system can monitor the events of join and leave produced at DHT layer and maintain automatically the list of keys and timestamps on P2P-LTR layer.

Once the stabilization of the DHT is completed, P2P-LTR assures that the old Master-key transfers its timestamps (if any) to the new Master-key. Let p and q be two peers, K the set of keys for which q is the current Master-key and p is a new peer that joins the system and becomes Master-key for some keys $K' \subseteq K$. Once q reaches the end of its responsibility for the keys involved in K' (i.e. when p joins the system), it first sends to p all the timestamps that have been generated for the keys involved in K' and then modifies its role towards these keys from Master-key to Master-key-Succ.

Let L be an empty set of pairs (key, timestamp), q performs the following instructions at the end of its responsibility:

for each $k \in K$ *do*
    if $(k \in K')$ *then*
        $L.add(k, t_k)$;
Send L to p;

Before receiving the timestamps from q, the peer p generates no timestamp for the keys involved in K'. So, P2P-LTR produces timestamps only if the DHT is stabilized. Therefore, any publish patch request received at the time of stabilization is rejected, while sending a message to the requester indicating that the system is executing the stabilization procedure.

As an illustration of a new Master-key peer join, consider the configuration of the P2P-LTR system shown in Figure 2. Assume that new peer P_0 joins the system and becomes the new Master-key peer of the key k_2. Figure 3 shows the new configuration of the system after the join operation, where P_2 transfers k_2 and t_2 to P_0. Consequently the new configuration of the peers P_0 and P_2 becomes:

- P_0 becomes Master-key of k_2 and Master-key-Succ of k_1
- P_2 becomes Master-key-Succ of key k_1

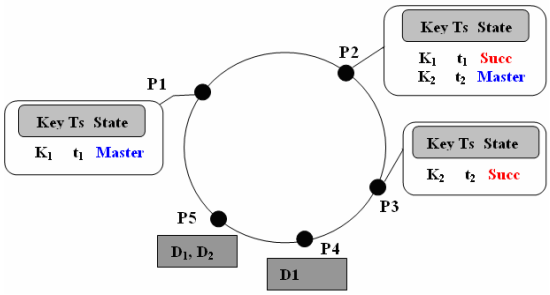

Figure 2. Initial configuration of P2P-LTR

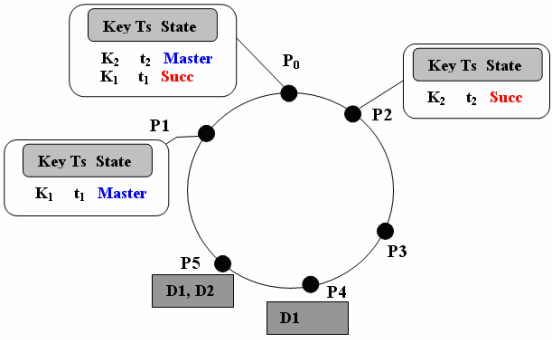

Figure 3. Configuration after the join of P_0

2) Departure

When a Master-key peer leaves the system normally, it transfers its keys and timestamps to its Master-key-Succ peer. Notice that, like the join event, the departure of a node triggers the DHT stabilization. Therefore, using the event interrupter, P2P-LTR can detect the departure event. Once the stabilization is completed, the set of keys and timestamp values of the previous Master-key are sent to its successor that becomes the new Master-key peer and to another peer that becomes the new Master-key-Succ. Let q and p be two peers, and K' be the set of keys for which q is the current Master-key and p is the Master-key-Succ, i.e. the next responsible of timestamping. Once q reaches the end of its responsibility for the keys in K', i.e. when it leaves the system, it sends to p all the timestamps that have been generated for the keys involved in K'. Consequently, p modifies its role towards the keys from Master-key-Succ to Master-key and sends these keys with its timestamps to its successor. Let $Role_{p,k}$ be the role of the peer p towards the key k. The peer p performs the following instructions:

for each k ∈ K' do
Rolep,k = Master;
Send (k,k.timestamp) to p.successor;
end;

Before receiving the timestamps from q, the peer p generates no timestamp for the keys involved in K'.

As an illustration of a Master-key peer departure, consider the initial configuration of the P2P-LTR system shown in Figure 2. Assume that P_1 leaves the system normally and notifies P_2 which becomes the new Master-key peer of the key k_1. Figure 4 shows the new configuration of the system after the leave operation, where P_2 modifies its role towards k_1 from Master-key-Succ to Master-key and replicates k_2 and t_2 to P_3 (i.e. its successor). Consequently the new configuration of the peers P_2 and P_3 becomes:

- P_2 becomes Master-key of k_1 and k_2
- P_3 becomes Master-key-Succ of key k_1 and k_2

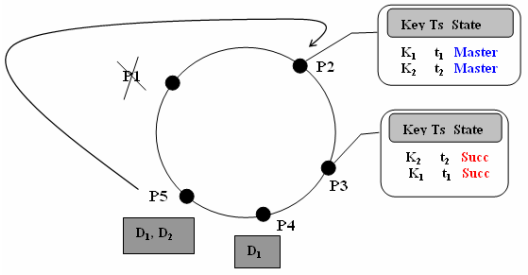

Figure 4. Handling Master-key peer departure

B. Failure Handling

Let us now study the effect of peer failures on P2P-LTR and discuss how they are handled. With peer failure, we mean that a peer disappears from the system without notifying other nodes in the system, *e.g.* it failures. We show that no peer failure can block our publish patch algorithm. We also show that even in the presence of these failures, P2P-LTR guarantees continuous timestamping. For this, it is sufficient to show that each generated timestamp is stored together with a patch in the DHT before generating the next timestamp or it is aborted.

1) Assumptions

Let us first describe the assumptions that we make in our work.

- *Assumption 1. (failure detectors):* In order to detect Master-key failures, we assume the existence of failure detectors [5] which can be installed at each peer in order to monitor the Master-key peer.
- *Assumption 2. (log-peer availability):* Let $T = [t_1, t_2]$ be a time interval such that t_1 and t_2 are the times at which the system starts and ends respectively. At any time $t \in T$ at least one of the peers, which are responsible for holding the patch, is available in the system. Formally, let D be a document, and n_t the number of the peers that are available at time t and hold a valid patch for D. Then, we assume that: $\forall t \in T \Rightarrow (n_t \geq 1)$
- *Assumption 3. (reliable message):* We assume the presence of reliable message transmission between different peers of the system.
- *Assumption 4. (correct lookup service):* Like several other protocols and applications designed over DHTs, e.g. [4], in our work we assume that the lookup service of the DHT behaves properly. That is, given a key k it either finds correctly the responsible for k or reports an error, *e.g.* in the case of network partitioning where the responsible is not reachable.

2) Dealing with the Failure of Master-key

To make P2P-LTR resilient to Master-key failures, we modify our patch distribution algorithm, which we proposed in Section III, as follows. After receiving the publish patch

request from a peer *p* and checking the timestamp value, the Master-key peer replicates the patch in the log-peers and waits for a confirmation message from log peers. Whenever a log-peer receives a patch from the Master-key, it sends a reliable message to the Master-key peer to confirm the reception of the replica. If at least *N* confirmation messages are received by the Master-key peer *(while* $N < n_t$ is a system parameter*)*, it sends an *ack* message to peer *p*, *i.e.* the publish patch requester, in order to validate the tentative patches. Otherwise, it sends an abort message to *p*, to indicate that the operation was not done successfully, so *p* should try again.

The behavior of the Master-key peer can be modeled by the Finite State Machine (FSM) [7] shown in Figure 5. A state machine is a model of system behavior composed of a finite number of states, transitions between those states, and actions.

Below, we discuss how P2P-LTR tolerates the failures in each state of this machine. A failure of a Master-key may happen in one of the following states:

- **Failure in the patch reception state (S1)**. In this case, the Master-key peer fails just after receiving the publish patch request. Since we use a failure detector at requesting peer *p*, it detects the failure of the Master-key peer, and the publish request is aborted. Similarly, the failure detector at the Master-key-Succ detects the failure and then that peer becomes the new Master-key. Therefore, in the case of failure in this state, the protocol does not block and continuous timestamping is assured, *i.e.* because no publish patch tentative is performed.
- **Failure in the timestamp generation state (S2)**. In this state, like in the previous one, the failure detector at requesting peer *p* detects the failure of the Master-key peer, and the publish request is aborted. Similarly, the failure detector at the Master-key-Succ detects the failure and then that peer becomes the new Master-key. Therefore, as in the previous state, continuous timestamping is assured.
- **Failure in the patch replication state (S3).** If the Master-key peer fails in this state, a fail message is sent to peer *p* to indicate the fail of publish patch tentative. Since the fault detectors installed at Log-Peers detect the fail of the Master-key, they simply discard the received patch. After a while, the successor of the former Master-key becomes Master-key peer. In this state, like in the previous one, the protocol does not block and continuous timestamping is assured, *i.e.* because the timestamp which is generated by the failed responsible, is aborted.
- **Failure in the last_ts replication state (S4)**. If the Master-key fails in this state, a fail message is sent to the peer *p* to indicate the failure of Master-key. The successor of the former Master-key becomes the new Master-key of the document. Using the set of replication hash functions, the new Master-key sends a message to the Log-Peers to ask them whether they received the patch. If at least *N* positive responses are returned, the Master-key-Succ sends an *ack* message to *p* to indicate the success of the publish patch operation. Otherwise, it sends an *abort* message to *p*. In this state, as in the previous state, continuous timestamping is assured.

Lemma 1. If during the execution of our protocol the Master-key peers fail, the protocol does not block and guarantees continuous timestamping.

Proof. Implied by the above discussion.

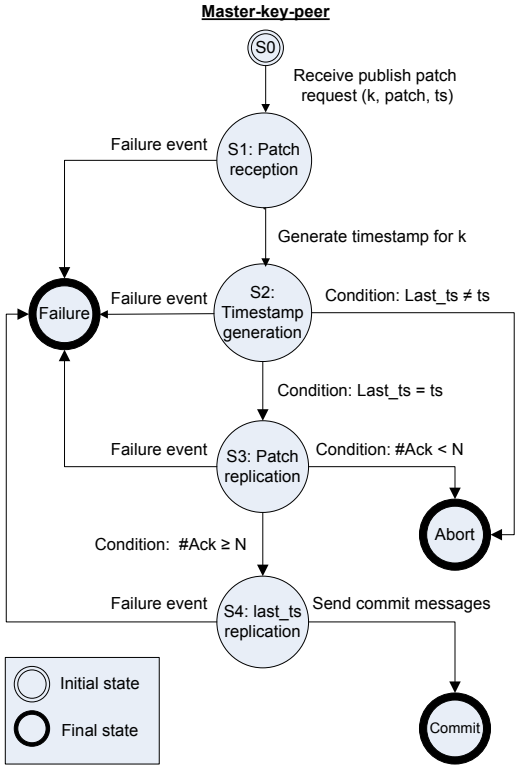

Figure 5. Master-key peer state transition machine

V. PERFORMANCE EVALUATION

In this section, we evaluate the performance of our P2P-LTR system through simulation using the PeerSim simulator [18]. This section is organized as follows. First, we describe our simulation setup, the metrics used for performance evaluation and the baseline service used for comparison with P2P-LTR. Then, we study the effect of the average number of users on the performance of P2P-LTR, and show how it scales up. Next, we study the effect of the number of replicas and the frequency of updates on the performance of P2P-LTR. Then, we investigate the effect of peer failures on the

correctness of P2P-LTR. Finally, we summarize the main results.

A. Simulation Setup

Our simulation is based on Chord which is a simple and efficient DHT. Chord's lookup service [23] is robust in the face of frequent node failures, and it can answer queries even though the system is continuously changing.

We implemented our simulation using the PeerSim simulator [18]. PeerSim is a Java based simulator specifically tailored for P2P protocol simulations. It consists of configurable components. It has two types of engines, cycle-based and event-driven. It provides different modules that manage the overlay building process and the transport characteristics.

Our simulation parameters are shown in Table 1. The latency between any two peers is a normally distributed random number with a mean of 200 ms. The simulator allows us to perform tests up to 10 000 peers. In our experiments, at each time $T = 30$ second, each user selects one document who modifies and tries to publish patches on the DHT. The default number of shared documents for each user is 10. The default average number of users who work on the same document is *5*. The default average rate for publish patch events is $\alpha = 1$ every *2mn*. In our tests, the number of replicas of each data is *10, i.e.* $|H_r|=10$.

TABLE I. EXPERIMENTAL PARAMETERS

Parameters	Values
Latency	Normally distributed random number, Mean = 200 ms, Variance = 100
Network size	10 000 peers
Number of Log peers for each data	10
Number of shared documents for each user	10
Update and publish patch generated on each replica	Timed by an uniform random process, Mean = 2 per minute
Fails rate	2%

In our tests, we compare the performance of P2P-LTR with So6*, a variant of So6 [14] that replicates the shared data in the DHT to improve data availability. So6* uses a central timestamper to determine the total order of the operations done on the data.

In our tests, we measured the performance of publishing and retrieving patches in terms of response time. By response time, we mean the time needed to publish the patch generated after an update.

B. Scalability

In this section, we study the scalability of P2P-LTR. For this, we study the effect of the average number of users per document on performance of P2P-LTR.

Figure 6 shows the response time of the patch publish operation with the average number of users increasing up to 30 users per document and the other simulation parameters set in Table 1. The average number of users working on the same document has a very slight impact on the response time of P2P-LTR which means an excellent scalability *w.r.t.* the numbers of users. When the number of users per document is more than 15, the response time of P2P-LTR is better than So6*. The higher is the number of users per document, the more is the factor by which P2P-LTR outperforms So6*. The reason is that with So6*, the timestamp generation and patch publication for all documents are performed by only one peer. In contrast, with P2P-LTR the responsibility of timestamp generation and patch storage are completely distributed over different peers of the DHT.

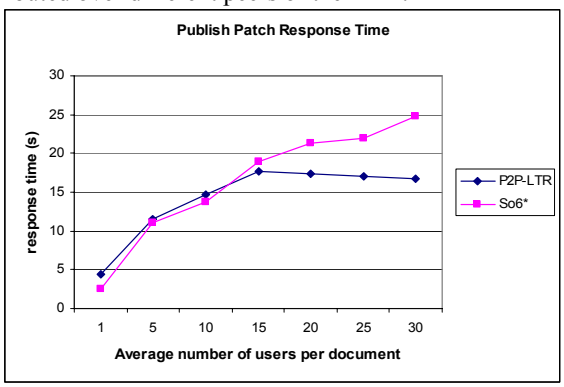

Figure 6. Response time vs. number of users.

Using the simulator, Figure 7 depicts the total number of messages while increasing the number of user peers per document up to 300, with the other simulation parameters set as in Table 1. The results show that the communication cost for P2P-LTR and So6* increases linearly with the number of user peers. However, the communication cost of P2P-LTR is a little bit higher than that of So6*. The reason is that P2P-LTR performs multiple lookups in the DHT for finding the Master-key of each document. Notice that each lookup needs *O(log N)* messages where *N* is the number of peers of the DHT. This slight increase in communication cost of P2P-LTR is the price to pay for guaranteeing continuous timestamping and making our solution more resilient to the failure cases.

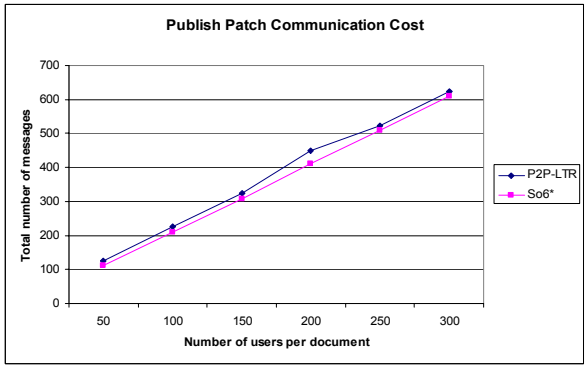

Figure 7. Total number of messages vs. number of users per document.

C. Effect of the Number of Replicas

In dynamic systems, we need many replicas to guarantee high data availability. We now study the effect of the number

of replicas, which we replicate for each patch in the DHT (in the Log-peers), on the performance of P2P-LTR and So6*.

Using our simulator, we studied how response time evolves while increasing the number of replicas, with the number of users per document is 15 and with the other simulation parameters set as in Table 1. The results (see Figure 8) show that increasing the number of replicas for P2P-LTR and So6* decreases the response time for publishing patches. The reason is that if we increase the number of replicas the average time for finding the missed patches decreases. Thus, the response time of the publish patch operation decreases. However, the response time of P2P-LRR is better than So6*. As explained before, the reason is that with So6*, the timestamp generation and patch publication for all documents are performed by only one peer.

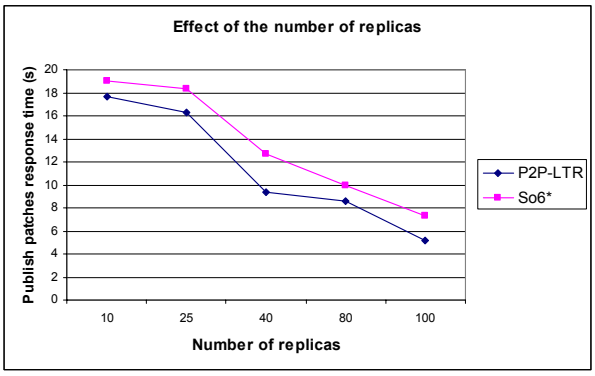

Figure 8. Response time VS number of log-peers.

D. *Effect of Update Frequency on Response Time*

In this section, we study the effect of the frequency of updates on the performance of P2P-LTR and So6*. In the previous experiments, updates on each data were timed by a random process with an average rate of 2 updates per minute. In this section, we vary the average rate (i.e. frequency of updates) and investigate its effect on response time.

Using our simulator, Figures 9 shows how response time evolves while increasing the frequency of updates with the other simulation parameters set as in Table 1. The results show that the response time of P2P-LTR decreases by increasing the frequency of updates. The reason is that an increase in the frequency of updates decreases the distance between the time of the current and latest update, thus number of missing patches decreases, and this decreases the time needed for retrieving the missing patches.

The response time of P2P-LTR is significantly better than So6*. The reason is that with So6*, the operations of timestamp generation and storing of timestamped patches for all documents are performed by only one peer. Indeed, by increasing the frequency of updates the central timestamper takes important time to generate timestamp and to publish the patches in the DHT.

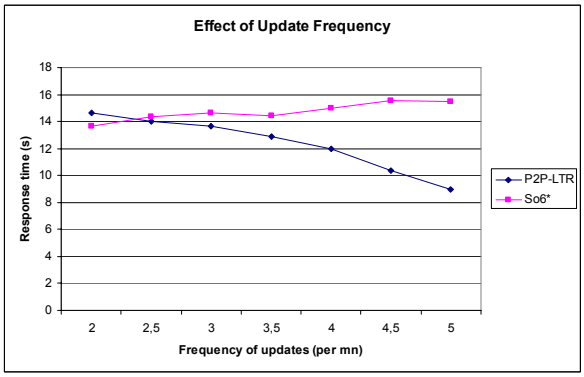

Figure 9. Publish patches time VS Frequency of updates.

E. *Effect of Peer Failures on Timestamps Continuity*

Let us now study the effect of peer failures on the continuity of timestamps used for data updates. In our experiments we measure *timestamp continuity rate* by which we mean the percentage of the updates whose timestamps are only one unit higher than that of their precedent update. We varied the fail rate parameter, and observed its effect on timestamp continuity rate.

Figure 10 shows timestamp continuity rate for P2P-LTR and So6* while increasing the fail rate, with the other parameters set as in Table 1. The peer failures do not have any negative impact on the continuity of timestamps used by P2P-LTR, because our protocol assures timestamp continuity. However, when increasing the fail rate in So6*, the percentage of updates whose timestamps are not continuous increases. In other words, in the presence of peer failures, there are gaps in the timestamps used by So6*, so it is not suitable for applications that require continuous timestamps.

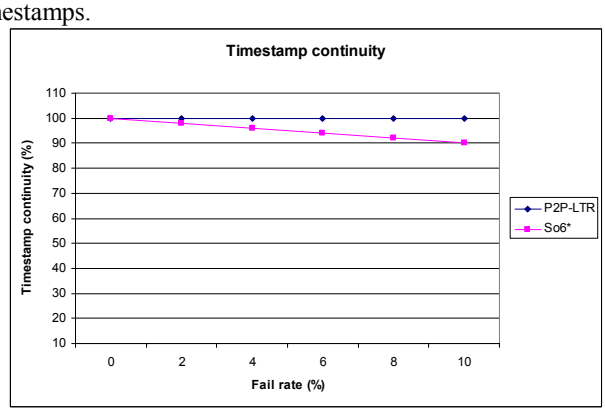

Figure 10. Timestamp continuity vs. fail rate

F. *Concluding Remarks*

In the previous sections, we evaluated the performance of P2P-LTR according to the following parameters: the average number of users, the number of replicas, the effect of update frequency and the effect of failures. In order to evaluate the response time (i.e. the time needed to publish a patch), we

compared P2P-LTR results with the So6* results (variant of So6 solution [14]).

The results show good response time of P2P-LTR since it outperforms the So6* solution by a factor around 1.5 when 30 users concurrently working on the same document (see Figure 6).

The results (see Figure 7) show also that the publish patch communication cost of P2P-LTR is linear to the number of users who working on the same document. However, the communication cost of P2P-LTR is a little bit higher than that of So6*. The reason is that P2P-LTR performs multiple lookups in the DHT for finding the Master-key of each document. Notice that this slight increase in communication cost of P2P-LTR is the price to pay for guaranteeing continuous timestamping and making our solution more resilient to the failure cases.

In figure 8, the results show that the performance of P2P-LTR is very good up to 80 Log-peers. However, the number of Log-peers can not grow indefinitely due to communication overhead. In the figure 9, we investigated the effect of frequency of updates on the performance of P2P-LTR and So6*. The results show that good performance of P2P-LTR since it outperforms the So6* solution when the frequency of updates up 2.5 updates per minutes. In addition, Unlike to So6*, the results show that the response time of P2P-LTR decreases by increasing the frequency of updates.

The latest results (see Figure 10) show that our solution works correctly even in the presence of peers failures and guarantees the timestamp continuity property. However, when increasing the fail rate in So6*, the percentage of updates whose timestamps are not continuous increases. Therefore, it is not suitable for applications that require continuous timestamps.

VI. Related Work

Guaranteeing eventual consistency in the presence of multimaster replication is a widely researched field. Many solutions have been proposed in the context of distributed database systems for managing replica consistency [17], in particular, using eager or lazy (multimaster) replication techniques. However, these techniques either do not scale up to large numbers of peers or raise open problems, such as replica reconciliation, to deal with the open and dynamic nature of P2P systems.

Distributed version control systems (DVCS) allow many users to edit the same documents concurrently. They provide the same features as CVS [6] without requiring a central site. DVCS do not support the consistency of replicas [3]. There are well-known scenarios in which most of DVCS (e.g. Git [9]) can not assure the consistency of replicas, see details in [3].

OceanStore [11] is a data management system designed to provide a highly available storage utility on top of P2P systems. It allows concurrent updates on replicated data, and relies on reconciliation to assure data consistency. The reconciliation is done by a set of powerful servers using a consensus algorithm. The servers agree on which operations to apply, and in what order. In the applications, which we address, the presence of powerful servers is not guaranteed. In our approach, for each data there is an ordinary peer, i.e. Master-key peer, that deals with replication management of the data, and the Master-key is determined dynamically using a hash function. Thus, the load of providing data consistency is distributed over all the peers of the system. In addition, we use the approach of timestamping which is less expensive than consensus.

Telex [2] is a P2P semantic reconciliation system designed for data sharing by distributed collaborative applications. Users operate on their local persistent replica of shared documents; they can work disconnected and suffer no network latency. To detect and correct conflicts, Telex sites maintain an Action-Constraint Graph (ACG) [22], i.e. a replicated dynamic graph that summarises the concurrency semantics of applications. However, the Telex system has a main limitation: it suffers from excessive memory consumption. The ACG can quickly reach sizes of several tens of thousands of nodes, and is accessed concurrently by many threads.

Collaborative editing based on the operational transformation approach (OT) [10][19] can be done in a decentralized way. There are several algorithms, e.g. GOTO [24], SOCT2 [25] and SOCT4 [25], for defining operation transformations and detecting concurrent operations. Based on these algorithms, many OT services have been developed. One of the well known OT services is So6 which is based on SOCT4. Although OT services are widely used in collaborative editing systems, several of them such as So6 rely on timestamps generated by a central stamper or a vector clocks, thus not appropriate for large scale P2P systems.

Woot [16] is a distributed merge algorithm recently designed for ensuring the convergence among replicas in P2P systems. Each site merges its own copy as soon as operations are received. The result is independent of the reception order. It requires no vector clocks, contrary to most operational transformation based algorithms. It is a specialized algorithm for linear structures like strings, so it is well suited for a wiki application that mainly manages page contents as text. However, Woot has its own specific data model keeping the history of all the operations brought to the data. Woot manages only linear structures and defines specific operation profiles. Unlike Woot, P2P-LTR offers the advantage of not requiring any specific storage. Moreover, P2P-LTR is generic and independent of data types.

In [26] we present our basic prototype used to demonstrate P2P-LTR through several scenarios. In the present paper, we describe in details all relevant algorithms necessary for patch validation face to peers dynamic behavior. In addition we studied the performance evaluation of P2P-LTR.

VII. Conclusion

In this paper, we addressed the problem of optimistic replication for P2P collaborative text editing. This problem is challenging because of concurrent updating at multiple peers and dynamic behavior of peers. Our solution is a scalable P2P logging and timestamping infrastructure called P2P-LTR which exploits a distributed hash table (DHT). While updating multimaster copies at collaborating peer editors,

updates are stored in a highly available P2P log. P2P-LTR provides an efficient mechanism for determining the total order of updates in order to enforce strong consistency and ensure liveness property. It also deals with the case of peers that may join and leave the system during the update operation.

We evaluated the performance of P2P-LTR through simulation. We compared P2P-LTR with a baseline service, i.e. So6*. When the number of users per document is more than 15, the results show that the response time for publishing a patch with P2P-LTR is significantly better than So6*. We also studied the effect of frequency of updates on the response time of patch published by P2P-LTR and So6*: The results show that the response time of P2P-LTR decreases with increasing the frequency of updates, so it more appropriate than So6* for large scale systems in which the frequency of updates is high. We also investigated the effect of peer failures on the correctness of P2P-LTR and So6*, the results show that P2P-LTR works correctly even in the presence of pee failures: P2P-LTR guarantees continuous timestamping. However, in the presence of peer failures, there are gaps in the timestamps used by So6*, so it is not suitable for applications that require continuous timestamps.

P2P-LTR is now an open source software, available at http://p2pltr.gforge.inria.fr/.

REFERENCES

[1] R. Akbarinia, E. Pacitti, P. Valduriez. Data Currency in Replicated DHTs. *ACM SIGMOD Int. Conf. on Management of Data*, 211-222, 2007.

[2] L. Benmouffok, J. Busca, J. Marquès, M. Shapiro, P. Sutra, G. Tsoukalas. Telex: Principled System Support for Write-Sharing in Collaborative Applications. Research Report RR6546, INRIA, 2008.

[3] G. Canals, P. Molli, J. Maire, S. Laurière, E. Pacitti, M. Tlili : XWiki Concerto : A P2P Wiki System Supporting Disconnected Work. *Int. Conf. on Cooperative Design, Visualization and Engineering (CDVE),* 98-106, 2008.

[4] Y. Chawathe, S. Ramabhadran, S. Ratnasamy, A. LaMarca, S. Shenker, J.M. Hellerstein. A case study in building layered DHT applications. *ACM SIGCOMM Conf*, 97-108, 2005.

[5] G. Chockler, I. Keidar and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4), 427-469, 2001.

[6] CVS: http://www.cvshome.org/docs/manual.

[7] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines: A survey. *In Proceedings of the IEEE*, 84(8):1090-1126, 1996.

[8] FreePastry: http://freepastry.org/FreePastry/

[9] Git. http://git-scm.com/

[10] A. Imine, P. Molli, G. Oster, and P. Urso, "Vote: Group editors analyzing tool: System description." *Electr. Notes Theor. Comput. Sci*, 86(1), 153-161, 2003.

[11] J. Kubiatowicz, et al. OceanStore: An Architecture for Global-Scale Persistent Storage. *Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 190-201, 2000.

[12] P. Linga et. Al.: Guaranteeing Correctness and Availability in P2P Range Indices, *ACM SIGMOD Int. Conf. on Management of Data*, 323-334, 2005.

[13] V. Martins and E. Pacitti. Dynamic and distributed reconciliation in P2P-DHT networks. *European Conf. on Parallel Computing (Euro-Par)*, 337-349, 2006.

[14] P. Molli, and. Al. Using the transformational approach to build a safe and generic data synchronizer. *ACM SIGGROUP Conf. on Supporting Group Work,* 212-220, 2003.

[15] Open Chord version 1.0.2 User's Manual. http://www.uni-bamberg.de

[16] G. Oster, P. Urso, P. Molli, H. Skaf-Molli and A. Imine. Optimistic Replication for Massive Collaborative Editing. Research Report RR-5719, INRIA, 2005.

[17] T. Özsu, P. Valduriez. Principles of Distributed Database Systems. *2nd Edition, Prentice Hall*, 1999.

[18] PeerSim simulator. http://peersim.sourceforge.net/

[19] M. Ressel, D. Nitsche-Ruhland and R. Gunzenhäuser. An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors. *ACM Conf. on Computer supported cooperative work (CSCW)*, 288-297, 1996.

[20] A. Rowstron, P. Druschel. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. *Middleware Conf.*, 329-350, 2001.

[21] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 42-81, 2005.

[22] M. Shapiro, K. Bhargavan, and N. Krishna. A constraint-based formalism for consistency in replicated systems. *In Int. Conf. on Principles of Dist. Sys. (OPODIS), number 3544 in Lecture Notes in Comp. Sc*, 331-345, 2004.

[23] I. Stoica and. Al. Chord: a scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Conf,* 149-160, 2001.

[24] C. Sun and C. Ellis. Operational Transformation in Real-Time Group Editors: Issues, Algorithms and Achievements. *In Proceedings of the ACM Conf. on Computer supported cooperative work*, 59-68, 1998.

[25] M. Suleiman, M. Cart, and J. Ferrié. Serialization of Concurrent Operations in a Distributed Collaborative Environment. *ACM SIGGROUP Conf. on Supporting Group Work*, 435-445, 1997.

[26] M. Tlili and. Al. P2P Logging and Timestamping for Reconciliation. *In Conf. on Very Large Data Bases VLDB*, (Demonstration session), 1420-1423, 2008.

[27] XWiki Concerto: http://concerto.xwiki.com.